A THIRD ORDER RUNGE-KUTTA ALGORITHM ON A MANIFOLD

P. E. Crouch, R. G. Grossman and Y. Yan

Center for Systems Science and Engineering
Arizona State University

Laboratory for Advanced Computing
University of Illinois at Chicago

December, 1991; Revised March, 1992

ABSTRACT

We describe a third order Runge-Kutta type algorithm with the property that it preserves certain geometric structures. In particular, if the algorithm is initialized on a Lie group, then the resulting iterates remain on the Lie group.

1. INTRODUCTION

The the main purpose of this paper is to describe a class of Runge-Kutta type algorithms with the property that they preserve certain geometric structures. We give some numerical results for a particular third order algorithm of this type. To explain the basic idea, we first describe how the algorithm would apply to the situation of integrating a differential equation on a finite dimensional Lie group G. Let $L(G)$ denote the Lie algebra of G and let $Y_1, \ldots, Y_N$ denote a basis of $L(G)$, identified as left (or right) invariant vector fields on G. We wish to numerically integrate solutions of differential equations evolving on G of the form:

$$\dot{x}(t) = F(x(t)), \qquad x(0) = p \in G, \tag{1}$$

where

$$F(x) = \sum_{\mu=1}^{N} a^{\mu}(x) Y_{\mu}(x), \qquad a^{\mu} \in C^{\infty}(G). \tag{2}$$

We say that such an algorithm is *geometrically stable* if $x^n \in G$ implies that $x^{n+1} \in G$. In order to apply traditional algorithms to this situation, one can embed G as a submanifold of a Cartesian space. In general, due to approximation errors, traditional algorithms are not geometrically stable, in the sense that although the initial value of the iteration scheme may lie exactly on the submanifold defined by G, subsequent iterates drift off the submanifold. One can modify such a scheme by projecting the iterates x^n back onto the group before computing the next approximation x^{n+1}. The algorithms we consider here are not of this type; rather we assume:

Assumption 1. There is an oracle which exactly computes the exponential map:

$$exp : L(G) \to G.$$

It turns out that this provides enough structure to yield a natural class of geometrically stable integration algorithms.

Remark 1.1 The algorithms we consider here have the property that if G is the abelian group R^N, then they reduce to classical Runge-Kutta algorithms.

Remark 1.2 For some groups, such as $SO(3)$, the exponential map can be written in closed form (see section 3 below); while for other groups, no such closed form expressions are known. However, in these latter cases we can approximate the exponential map to any desired accuracy (even off line) and, in particular, to an accuracy independent of the order of the underlying algorithm, computed under Assumption 1.

Remark 1.3 Since elements of the Lie algebra may be viewed as invariant vector fields on the group, their values at one point, say at the identity, determine their values throughout the group. Assumption 1 therefore implies that we can exactly solve (or solve to any prescribed accuracy) the "constant coefficient" differential equations obtained from equations (1) and (2) by substituting constants for the coefficient functions a^μ. The idea of solving a general differential equation by "freezing" the coefficients at a point to get a "constant coefficient" differential equation is a very old one.

Remark 1.4 If M is a homogeneous space so that M may be identified with a quotient of two Lie groups, G/H, then we may consider G as a subset of the diffeomorphism group of M. Thus, G acts on M by differentiable mappings $(g, x) \to g \cdot x$ and every element of M may be expressed in the form $g \cdot x$, for some fixed $x \in M$ and some $g \in G$. We may now consider a slightly more general version of the equations (1) and (2), evolving on a homogeneous space M:

$$\dot{x}(t) = F(x(t)), \qquad x(0) = p \in M, \tag{3}$$

where

$$F(x) = \sum_{\mu=1}^{N} a^\mu(x) Y_\mu^*(x), \qquad a^\mu \in C^\infty(M) \tag{4}$$

and

$$Y_\mu^*(x) = \frac{d}{dt}(exp tY_\mu \cdot x)|_{t=0}, \qquad x \in M.$$

Here Y_μ are elements of $L(G)$ described above. Denoting the flow of an arbitrary vector field F on M also by "exp," $(t, x) \to exp tF(x)$, it is well known that we have:

$$exp tY_\mu^*(x) = exp tY_\mu \cdot x, \qquad x \in M. \tag{5}$$

It is easy to extend our algorithms to problems defined by the equations (3) and (4), since by equation (5), the differential equation obtained from equation (3) by freezing the coefficient functions a^μ has solutions that are also determined by the oracle in assumption 1, and by the evaluation of quantities $g \cdot x$, $x \in M$ and $g \in G$. A simple class of examples that are captured by this more general situation are those in which M is one of the spheres S^n. In section 4 we give results of numerical experiments where we compare our algorithm with other classical algorithms for the case where M is the two sphere.

Remark 1.5 Our algorithms generalize from differential equations evolving on Lie groups and homogeneous spaces to more general manifolds. However, in this case we must replace the oracle in assumption 1 by a similar object, which is able to compute exact solutions of the corresponding differential equations with frozen coefficients.

In section 2 we describe the class of algorithms. In section 3 we define a computation model and measure the complexity of the algorithm with respect to this model. Section 4 contains some numerical studies and Section 5 contains some concluding remarks.

2. Description of the Algorithms

Consider a differential equation (3), on a manifold M, with initial condition p and defined by a vector field F having the form:

$$F(x) = \sum_{\mu=1}^{N} a^{\mu}(x) A_{\mu}(x), \qquad x \in M \tag{6}$$

where A_μ are vector fields on M and a^μ are coefficient functions on M. Then, given any real valued function ϕ on M, we may expand the solution $x(h)$ in a Taylor series about $h = 0$, in the following way:

$$\begin{aligned} &\phi(x(h)) = \phi(exp\, hF(p)) = \\ &\phi(p) + hF(\phi)(p) + \frac{h^2}{2!}F(F(\phi))(p) + \frac{h^3}{3!}F(F(F(\phi)))(p) + \cdots \end{aligned} \tag{7}$$

where $F(\phi)$ denotes the Lie derivative of ϕ by F. The new class of algorithm we introduce here is obtained by comparing this Taylor series with the Taylor series about $h = 0$, obtained from the following expressions:

$$\phi(exp\, hc_k F_k^p(exp\, hc_{k-1} F_{k-1}^p(\ldots(exp\, hc_1 F_1^p(p))\ldots))) \tag{8}$$

where

$$\begin{aligned} F_1^p(x) &= \sum_{\mu=1}^{N} a^{\mu}(p) A_{\mu}(x) \\ F_2^p(x) &= \sum_{\mu=1}^{N} a^{\mu}(exp\, hc_{21} F_1^p(p)) A_{\mu}(x) \\ F_3^p(x) &= \sum_{\mu=1}^{N} a^{\mu}(exp\, hc_{32} F_2^p(exp\, hc_{31} F_1^p(p))) A_{\mu}(x) \end{aligned} \tag{9}$$

... etc. The constants c_i and c_{ij}, $i = 1 \ldots k$, $j < i$ are determined as solutions of the equations obtained by insisting that the Taylor series agree up to and including terms in h^M, where M ($\leq k$) will be the *order* of the resulting algorithm.

Algorithm 1 (Adapted to Lie groups) To integrate the equations (1) and (2), we set $M = G$, $A_\mu = Y_\mu$ in equation (6) and obtain the algorithm:

$$x^{n+1} = exp\, hc_k F_k^{x^n} \cdot exp\, hc_{k-1} F_{k-1}^{x^n} \cdot \ldots \cdot exp\, hc_1 F_1^{x^n} \cdot x^n \tag{10}$$

where multiplication of two elements g_1, g_2 in G is denoted by $g_1 \cdot g_2$. Notice that if $x^n \in G$, then $x^{n+1} \in G$ as required for a geometrically stable algorithm. The vector fields:

$$F_j^{x^n}$$

are left invariant vector fields on G, obtained by freezing the coefficient functions a^μ of F at various points in G as demonstrated by the equations (9).

Algorithm 2 (Adapted to homogeneous spaces G/H) To integrate equations (3) and (4), we set $M = G/H$, $A_\mu = Y_\mu^*$ in equation (6) and obtain the algorithm:

$$x^{n+1} = exp\, hc_k F_k^{x^n} \cdot exp\, hc_{k-1} F_{k-1}^{x^n} \cdot \cdots \cdot exp\, hc_1 F_1^{x^n} \cdot x^n \tag{11}$$

where (by using equation (5)) we may assume that the vector fields

$$F_j^{x^n}$$

again lie in $L(G)$. Thus, if $x^n \in G/H$, then $x^{n+1} \in G/H$ also, as required for a geometrically stable algorithm.

Remark 2.1 The equations constraining the constants c_i and c_{ij}, $i = 1 \ldots k, j < i$ may be derived in different ways. The paper [3] uses the algebra of Cayley trees generated by labeled, ordered trees introduced in [7], [8] and [9] to derive the equations. The paper [4] derives the constraint equations via a careful geometric analysis of the equations (7), (8) and (9).

These results show that for third order algorithms, $M = 3$, we obtain multiple solutions of these equations for $k = M = 3$, just as in the case $G = R^N$. However, the analysis of fourth order case $M = 4$ is significantly more complicated and not complete at this time. We report on the performance of one such third order algorithm in section 4 by choosing one of the sets of solutions.

Remark 2.2 If $G = R^N$ and $Y_j = e_j$ is the standard jth basis vector in R^N, then

$$F_1^p = F(p), F_2^p = F(p + hc_{21}F_1^p), F_3^p = F(p + hc_{31}F_1^p + hc_{32}F_2^p) \tag{12}$$

etc. ..., and the update rule (10) becomes:

$$x^{n+1} = x^n + h(c_1F_1^{x^n} + c_2F_2^{x^n} + \cdots + c_kF_k^{x^n}). \tag{13}$$

Equations (12) and (13) are now in the standard form of an explicit classical Runge-Kutta algorithm. In this case we can always take k to be the order M of the algorithm.

3. A Computational Model

In this section we describe the numerical cost of the algorithm with respect to the following computational model:

(1) If $g_1, g_2 \in G$, then the cost of evaluating their product in G, $(g_1 \cdot g_2)$, is one *group evaluation* unit.

(2) If $g \in G$ and $x \in G/H$, then the cost of evaluating the product $g \cdot x$, is one *homogeneous space evaluation* unit.

(3) If $f : G \to R$ ($f : G/H \to R$) is a function on G (on G/H), then the cost of evaluating f at $g \in G$ (at $x \in G/H$), is one *function evaluation* unit.

(4) If $Y \in L(G)$, then the cost of evaluating $expY$ is one *exponential evaluation* unit.

(5) If

$$\alpha^\mu \in R, \qquad Y_\mu \in L(G)$$

then the cost of evaluating:

$$F = \sum_{\mu=1}^{N} \alpha^\mu Y_\mu \in L(G)$$

is one *vector multiply evaluation* unit.

(6) If $\alpha \in R$, $Y \in L(G)$, then the cost of evaluating $\alpha Y \in L(G)$, is one *scalar multiply evaluation* unit.

(7) If $a, b \in R$, then the cost of evaluating $a + b$, ab, $\sin a$, $\sqrt{a}$, etc., is one *arithmetic evaluation* unit.

Theorem 3.1 If G has dimension N, then *Algorithm* 1 (*equations* (9) *and* (10)) *on a Lie group* G *requires*:

(1) $k(k+1)/2$ *group evaluation units.*
(2) kN *function evaluation units.*
(3) $k(k+1)/2$ *scalar multiply evaluation units.*
(4) $k(k+1)/2$ *arithmetic evaluation units.*
(5) $k(k+1)/2$ *exponential evaluation units.*
(6) k *vector multiply evaluation units.*

Algorithm 2 (*equations* 9 *and* 11) *on a homogeneous space* G/H *requires the evaluation units* 2 *through* 6 *and*:

(7) $k(k+1)/2$ *homogeneous space evaluation units.*

Example 3.1 If $G = R^N$, so that equations (9) and (10) are replaced by equations (12) and (13), then items 5 and 6 in theorem 3.1 are not required in the algorithm, one group evaluation costs N arithmetic evaluation units and one scalar multiply also costs N arithmetic evaluation units. Thus, in this case, each step of the algorithm costs kN function evaluations and $k(k+1)(N+1/2)$ arithmetic operations. As noted before, in this case we may insist that the order of the algorithm is simply k.

Example 3.2 If $G = SO(3)$ is viewed as a subgroup of the 3×3 nonsingular matrices, then we may write the differential equations (1) and (2) in the form:

$$\dot{R} = S(\omega(R))R, \qquad R \in SO(3) \tag{14}$$

Here $S(.)$ is a 3×3 skew symmetric matrix, satisfying $S(a)b = b\times a$, where $\times$ is the cross product of vectors $a, b \in R^3$. Then oracle must solve the equation with frozen coefficients, $\omega(R) = \alpha$:

$$\dot{R} = S(\alpha)R, \qquad \alpha \in R^3, R \in SO(3). \tag{15}$$

This equation has an explicit solution:

$$R(t) = e^{S(\alpha)t}R(0) = e^{S(c)\phi t}R(0), \qquad \alpha = c\phi, \qquad c^T c = 1 \tag{16}$$

where

$$e^{S(c)\phi t} = (I_3 + S(c)\sin(\phi t) + S(c)^2(1 - \cos(\phi t))). \tag{17}$$

Using theorem 3.1, we may compute the cost of each step of algorithm 1 applied to the equation (14). Because of the simple structure of equation (14), there is no cost associated with the vector multiply operation; clearly group evaluation is simply matrix multiplication of 3×3 matrices; scalar multiply costs three arithmetic evaluation units; exponential evaluation involves the computation in equations (16) and (17) and function evaluation is simply the evaluation of $\omega(R)$. The total cost of each step of algorithm 1 is approximately $100(k(k+1)/2))$ arithmetic evaluation units and $3k$ function evaluation units.

We may compare the cost of integrating equation (14) using algorithm 1, with the cost of integrating the same equation when viewed simply as an equation evolving in R^9 and using a classical Runge-Kutta, as analyzed in example 3.1, with $N = 9$. The cost of each step of the classical algorithm is then $(27k + 19(k(k+1)/2))$ arithmetic evaluation units and $3k$ function evaluation units. Assuming that the step length and k are both equal, the geometrically exact algorithm 1 is more expensive, as expected.

Example 3.3 Since S^2 is the homogeneous space of $SO(3)$, we may write equations (3) and (4), for $M = S^2$ in the form:

$$\dot{r} = \omega(r) \times r, \qquad r \in R^3. \tag{18}$$

We solve the same equation with frozen coefficients, $\omega(r) = \alpha$, using the oracle for $SO(3)$ in equations (16) and (17).

Using theorem 3.1 we see that the cost of algorithm 2, applied to equation (18), can be calculated as in example 3.2, with the only significant difference being that each group evaluation unit is replaced by a homogeneous space evaluation unit which is the product of a 3×3 matrix with a three vector. Thus the cost of each step of algorithm 2 is approximately $70(k(k+1)/2)$ arithmetic evaluation units and $3k$ function evaluation units, where each function evaluation is simply the evaluation of $\omega(r)$. The cost of integrating equation (18) via a classical Runge-Kutta algorithm may be computed using example 3.1 as $(9k+19(k(k+1)/2))$ arithmetic evaluation units and $3k$ function evaluation units per step. Again, it is clear that algorithm 1 is more expensive, assuming that the step length and k are both equal.

4. Numerical Experiments

We based our numerical experiments on equation (18) and tested algorithm 2 on three functions $\omega(r)$ chosen at random, as given below: We note that in the case of ω_1 the equation (18) already has frozen coefficients relative to the structure of the homogeneous space S^2 as explained in example 3.3.

We integrated equation (18) using three schemes:

A. The classical "Kutta" algorithm, which is an example of a third order Runge-Kutta algorithm described in example 3.1, using the coefficients:

$$c_1 = 1/6, c_2 = 2/3, c_3 = 1/6,$$
$$c_{21} = 1/2, c_{31} = -1, c_{32} = 2.$$

B. The algorithm 2, with coefficients:

$$c_1 = 1, c_2 = -2/3, c_3 = 2/3,$$
$$c_{21} = -1/24, c_{31} = 161/24, c_{32} = -6.$$

C. The Runge-Kutta algorithm found in the I.M.S.L. package.

All numerical experiments were performed on a VAX 6000-420. Schemes A and B were implemented in Fortran in double precision. Three different step lengths were tested, $h = 0.01$, 0.05 and 0.1. The main test of the new algorithm 2 (scheme A) is the accuracy with which the algorithm maintains iterates on S^2. In our case we used an initial state:

$$r_1 = r_2 = r_3 = 1$$

so that an iterate r^n remains on S^2 as long as

$$e^n = (r_1^n)^2 + (r_2^n)^2 + (r_3^n)^2$$

satisfies $e^n = 3$.

In the following tables we tabulate the error $e^n - 3$ for the classical algorithm (scheme A) and the IMSL algorithm 3 (scheme C) for $h = 0.05$ and 0.1. *All the new algorithms (scheme* B*) and the* IMSL *algorithm for* $h = 0.01$ *gave zero error for all times up to* $t = 1000$ and hence have not been tabulated. Times for $t > 20$ in tables 4.2 and 4.3 have not been tabulated either, since in these cases the algorithms converge to an equilibrium of the equation.

Table 4.1 $e^n - 3$ for equation 18 with coefficient fns ω_1

Time t	A	A	A	C	C
	$h = 0.01$	$h = 0.05$	$h = 0.1$	$h = 0.05$	$h = 0.1$
0.5	-.35E-5	-.43E-3	-.33E-2	-.10E-9	-.17E-7
1	-.70E-5	-.86E-3	-.66E-2	-.30E-9	-.35E-7
2	-.14E-4	-.17E-2	-.13E-1	-.60E-9	-.69E-7
5	-.35E-4	-.43E-2	-.32E-1	-.15E-8	-.17E-6
10	-.70E-4	-.86E-2	-.62E-1	-.29E-8	-.35E-6
20	-.14E-3	-.17E-1	-.11E+0	-.58E-8	-.69E-6
50	-.35E-3	-.41E-1	-.23E+0	-.14E-7	-.17E-5
100	-.70E-3	-.78E-1	-.34E+0	-.29E-7	-.35E-5
200	-.14E-2	-.14E+0	-.41E+0	-.58E-7	-.69E-5
500	-.35E-2	-.27E+0	-.43E+0	-.15E-6	-.17E-4
1000	-.70E-2	-.37E+0	-.43E+0	-.29E-6	-.35E-4

Table 4.2 $e^n - 3$ for equation 18 with coefficient fns ω_2

Time t	A	A	A	C	C
	$h = 0.01$	$h = 0.05$	$h = 0.1$	$h = 0.05$	$h = 0.1$
0.5	-.41E-7	-.53E-5	-.44E-4	-.20E-9	-11E-7
1	-.14E-6	-.18E-4	-.15E-3	-.60E-9	-.34E-7
2	-.39E-6	-.50E-4	-.40E-3	-.70E-9	-.40E-7
5	-.62E-6	-.77E-4	-.60E-3	.30E-9	.26E-7
10	-.62E-6	-.77E-4	-.60E-3	.30E-9	.26E-7
20	-.62E-6	-.77E-4	-.60E-3	.30E-9	.26E-7

Table 4.3 $e^n - 3$ for equation 18 with coefficient fns ω_3

Time t	A	A	A	C	C
	$h = 0.01$	$h = 0.05$	$h = 0.1$	$h = 0.05$	$h = 0.1$
0.5	-.57E-6	-.59E-4	-.27E-3	.45E-7	.32E-5
1	-.66E-6	-.69E-4	-.34E-3	.48E-7	.35E-5
2	-.68E-6	-.71E-4	-.35E-3	.49E-7	.35E-5
5	-.68E-6	-.71E-4	-.35E-3	.49E-7	.35E-5
10	-.68E-6	-.71E-4	-.35E-3	.49E-7	.35E-5
20	-.68E-6	-.71E-4	-.35E-3	.49E-7	.35E-5

In the following tables we tabulate the differences between iterates of the first component, r_1, computed using the IMSL routine with $h = 0.01$, and the corresponding iterates computed using schemes A, B and C ($h = 0.05, 0.1$) and

Table 4.4 Difference between iterates of r_1, computed with the IMSL Runge-Kutta routine, $h = 0.01$ and the stated algorithm, for equation 18 with coefficient fns ω_1

	A	A	A	B	B	B	C	C
t/h	0.01	0.05	0.1	0.01	0.05	0.1	0.05	0.1
0.5	. 17E-5	.18E-3	.11E-2	.00E+0	.00E+0	.00E+0	-.50E-9	-.28E-7
1	.28E-5	.41E-3	.38E-2	.00E+0	.00E+0	.00E+0	.14E-8	.99E-7
2	.38E-6	-.11E-3	-.23E-2	.00E+0	.00E+0	.00E+0	-.27E-8	-.19E-6
5	-.25E-4	-.32E-2	-.25E-1	.00E+0	.00E+0	.00E+0	-.31E-8	-.27E-6
10	-.51E-4	-.64E-2	-.49E-1	.00E+0	.00E+0	.00E+0	-.42E-8	-.40E-6
20	-.10E-3	-.12E-1	-.88E-1	.00E+0	.00E+0	.00E+0	-.70E-9	-.25E-6

Table 4.5 Difference between iterates of r_1, computed with the IMSL Runge-Kutta routine, $h = 0.01$ and the stated algorithm, for equation 18 with coefficient fns ω_2

	A	A	A	B	B	B	C	C
t/h	0.01	0.05	0.1	0.01	0.05	0.1	0.05	0.1
0.5	-.59E-7	-.73E-5	-.57E-4	-.16E-6	-.19E-4	-.14E-3	.00E+0	.19E-8
1	-.20E-7	-.41E-5	-.49E-4	-.51E-7	-.15E-4	-.19E-3	.20E-9	.11E-7
2	-.29E-6	-.38E-4	-.33E-3	-.50E-6	-.71E-4	-.63E-3	.30E-9	.16E-7
5	-.53E-7	-.64E-5	-.48E-4	.56E-7	.80E-5	.73E-4	.10E-9	.46E-8
10	-.91E-7	-.11E-4	-.88E-4	.10E-9	.20E-7	.23E-6	.00E+0	.37E-8
20	-.91E-7	-.11E-4	-.88E-4	.00E+0	.00E+0	.00E+0	.00E+0	.38E-8

Table 4.6 Difference between iterates of r_1, computed with the IMSL Runge-Kutta routine, $h = 0.01$ and the stated algorithm, for equation 18 with coefficient fns ω_3

	A	A	A	B	B	B	C	C
t/h	0.01	0.05	0.1	0.01	0.05	0.1	0.05	0.1
0.5	-.64E-6	-.81E-4	-.62E-3	-.59E-6	-.10E-3	-.99E-3	.11E-7	.79E-6
1	-.19E-6	-.25E-4	-.20E-3	.62E-6	.58E-4	.17E-3	.11E-7	.78E-6
2	-.11E-6	-.97E-5	-.18E-4	-.11E-6	-.61E-5	.47E-4	.11E-7	.76E-6
5	-.15E-6	-.16E-4	-.77E-4	-.60E-9	-.15E-6	-.18E-5	.11E-7	.77E-6
10	-.15E-6	-.16E-4	-.77E-4	.00E+0	.00E+0	.00E+0	.11E-7	.77E-6
20	-.15E-6	-.16E-4	-.77E-4	.00E+0	.00E+0	.00E+0	.11E-7	.77E-6

5. Discussion

In this note, we have examined a class of geometically stable numerical integration algorithms. The price of geometric stability is an increased cost of computation, when compared to a classical algorithm of the same type, with the same number of stages and the same step length. The exact cost is examined in section 3. However, the numerical results in section 4 demonstrate that the new algorithms are capable of achieving performance comparable with the IMSL routine, (which is fifth order) at larger step lengths ($h = 0.05/0.1$) compared with $h = 0.01$. Thus, it seems that the new class of algorithms does indeed have desirable properties, making them worthy of further investigation.

In the work [4] the authors have described a similar class of geometrically stable multistep algorithms on Lie groups, G, which also have the property of degenerating to classical multistep counterparts in the case $G = R^N$.

References

[1] J. C. Butcher, "An order bound for Runge-Kutta methods," *SIAM J. Numerical Analysis,* **12** (1975), pp. 304–315.

[2] J. C. Butcher, *The Numerical Analysis of Ordinary Differential Equations,* John Wiley, 1986.

[3] P. Crouch, R. Grossman, and R. G. Larson, "Trees, bialgebras, and intrinsic numerical algorithms," *Laboratory for Advanced Computing Technical Report* Number LAC90-R23, University of Illinois at Chicago, May, 1990.

[4] P. E. Crouch and R. Grossman, "Numerical integration of ordinary differential equations on manifolds," *Laboratory for Advanced Computing Technical Report* Number LAC91-R3, University of Illinois at Chicago, 1991.

[5] P. Crouch, R. Grossman, and R. G. Larson, "Computations involving differential operators and their actions on functions," *Proceedings of 1991 International Symposium on Symbolic and Algebraic Computation,* ACM, 1991, pp. 301–307.

[6] R. Grossman, "Using trees to compute approximate solutions of ordinary differential equations exactly," *Computer Algebra and Differential Equations,* M. F. Singer, editor, Academic Press, New York, 1991, in press.

[7] R. Grossman and R. Larson, "Hopf algebraic structures of families of trees," *J. Algebra,* Vol. 26 (1989), pp. 184–210.

[8] R. Grossman and R. Larson, "Labeled trees and the efficient computation of derivations," in *Proceedings of 1989 International Symposium on Symbolic and Algebraic Computation,* ACM, 1989, pp. 74-80.

[9] R. Grossman and R. Larson, "The symbolic computation of derivations using labeled trees," *Journal of Symbolic Computation,* to appear.

[10] V. S. Varadarajan, "Lie Groups, Lie Algebras, and their Representations," Prentice-Hall, Inc., 1974.